# CMPT 376W Reading Coding and Git Commit Messages

By: Michael Chang Date: February 3rd, 2021

Coding & Style Guidelines

Technical Style

Learn to Write

- Balance all the writing with a healthy dose of reading
- Read through a number of pieces by the same author to see what makes that's person's writing distinctive
    - Ex. Malcolm Gladwell
        - His style is one that'd work very well for technical documentation
        - Breezy, fun, conversational tone
- Good writing is clear, succinct, and communicates ideas effectively

Grammar

- It helps us to communicate our thoughts without ambiguity or confusion.
- Recommended books
    - The Elements of Style by Strunk & White
    - A Pocket Style Manuel by Diana Hacker
    - Handbook of Technical Writing by Gerald J. Aired, Charles, T. Brusaw, and Walter, E. Oliu

Style

It tells you

- when to spell out numbers and
- when to write them as digits,
- Where to use em- and en-dashes
- How to cite sources

3 important style

1. Chicago Manual
    i. Popular in social science & historical settings
2. MLA
    i. Used by most literary, media, and cultural publications
3. APA
    i. Basis of most scientific style guides

Be consistent with your styling

Just choose one style and use it

Good documentation style

Markup

A good online documentation is a important markup style

- Use inline markup liberally
    - Use emphasis and strong text frequently
    - Use it for *code samples*, *entry* vs *output*
    - This breaks up a large chunk of text, and let users to skim between different parts of the document
- Write in short paragraph
    - A paragraph in a book might have around 7-10 sentences long, but a good online documentation is half of that
    - Break up our thoughts into small pieces
        - This allows reader to not missed important points
- Use a variety of structural elements
    - Academic & journalist don’t use feature lists, tables, code blocks

- Most style omit using them
- They are actually important
  - Tables & lists are important for presenting material

- Make your structure visual
  - Header are very important to get right
    - They'll stop skimmers as the fly by down the page
  - Headline help reader quickly find the section of the document they're looking for

Style

- Be conversational
  - Write with similar to how you talk
  - Don’t use "well, you see, umm, gonna" or throw away grammar rules
- Don't be afraid to strike a personal tone
  - It's okay to use "I" in technical writing
  - Either use "I" or "we", choose one of them and stick to it
- Do be careful with tenses & persons
  - Most documentation written in second person, but there's no correct way here
- Watch out for passivity
  - Avoid passive voice in your writing
  - Good technical writings are active

- Omit fluff
  - Avoid words like "pretty, most, good"
- Watch out for written tics
  - Avoid certain bad habits in writing
  - You need to find them first

You Need an editor

Editing tips

- Don’t edit without permission
  - People writing documentation always volunteers
  - Ask for permission b4 you dive in.
  - It's difficult to tell someone who made a writing mistake without sounding like a jerk
- Be Prepared
  - Make sure you've got your style guide
  - ALWAYS HAVE A QUESTION as you go
- Avoid editing your own work
- Edit on paper
- Read slowly:
  - Aim to take as long with each word as possible
- Make couple 3 passes
  - Sometimes you just have to move on

Self-editing

- Avoid editing & writing simultaneously
  - Finish writing first, and then edit
- Give it some time
  - Especially between writing and editing
- Change your margin
  - Change column width or margin in your editor

Well-formed Commit Messages

Model Git commit message

1. Capitalized, short (50 chars or less) summary
2. Wrap it to about 72 characters or so for more detailed explanation.
3. First line is treated as the subject of an email, and the rest are the body
4. Blank line separating the summary from the body is very important
5. Write commit message in the imperative
    - "fix bug" not "Fixed bug" and "Fixes bug"

More paragraphs come after blank lines

- Bullet points are okay, too
- Hyphen/asterisk is used for the bullet
- Use a hanging indent

Reason to wrap your commit messages to 72 columns

- Git log
    - doesn't do any special wrapping of the commit message.
- Git format-patch --stdout
    - Converts a series of commits to a series of emails

Other commands

- Git log --pretty=online
    - Shows terse history mapping containing the commit id & summary
- Git rebase --interactive
    - Provide the summary for each commit in the editor it invokes
- if merge summary is set, then summaries will make their way into the merge commit message
- Git shortlog
    - Use summary lines in the changelog-like output it produces
- Git format-patch, git send-email,
    - Use it as the subject for emails
- Git has column for the summary
- GitHub uses the summary in various places in their UI

Programming Feels Like 90% Documentation and 10% Coding

How to write great documentation

1. Separate marketing/sales material from technical guides
    i. When someone bought the software, stop selling & start educating

2. Identify your user workflows (and fail flows) and create task-based documentation around those
    i. You can't tell someone every functionality of buttons and menu if they don't know what they're supposed to be doing

3. Standardize on DITA
    i. It can deliver in multiple mediums with a minimum amount of work
    ii. We can add installation guide to new platform easily by reusing existing content pieces from our current docs and writing new top-level pieces specific to new platform
        1. Then, we can publish to HTML, Java Help, or PDF

4. Do not micromanage documentation project if you're not professional/editor
    i. Trust the experts
    ii. Most documentation product owner is very hands-off, which allows other expert to create superior doc that can reduce support case load.

5. Get you developers to do a brain dump when they're coding something new

i. Especially for those developers who hate writing
ii. Record a video demonstrating the new feature

Programming feels like 90% documentation

Programming
90% documentation by time involvement
70% reading documentation
20% writing/explaining
10% actual coding

Reading comprehension cannot be a programmer's weakness. In fact, to be a proficient coder, it must be one of your strengths.

The Golden Rules of Code Documentation
The Golden Rules of Code Documentation

Code is documentation
Code is the most important part of the documentation, and its for those who
- Don’t know the code (someone else wrote it)
- Don’t have time to read the code (its too complex)
- Don't want to read the code (who wants to read Hibernate/Xerces code to understand what's going on?)
- Don’t have access to the code (although they could still decompile it)

API is documentation
The code in API should be very simple and concise

Good design
- Don't let methods with more than 3 arguments leak into your public API.
- Don’t let methods/types with more than 3 words in their names leak into your public API.

API should be documented in words
- When the code is release to the public API, it should be documented in human-readable words

Tracking tools are documentation
Tracking tools - human interface to your stakeholders
- It'll help you discuss things
- Provide history of why your code is written like this

When your modifying your code, make sure to add comment to the relevent code section (like referencing the ticket ID)

You can also reference your ticket ID in:
- Mailing lists
- Source code
- API documentation
- Version control checkin comments
- Stack Overflow questions
- All sorts of other searchable documents
- etc

Version control is documentation

Documents change
Important to include the ticket ID

Follow this Rule

- Is the change non-trivial (fixed spelling, fixed indentation, renamed local variable, etc)
  - Then create a ticket and document this change with a ticket ID in your commit

Version numbering is documentation

Use [X].[Y].[Z] versioning scheme

- (patch release includes bugfixes, performance improvements & API-Irrelevant new features) => ([Z] is incremented by one)
- (a minor release includes backwards-compatible, API-relevant few feature) => ([Y] is incremented by one & reset to zero)
- (a major release includes backwards-incompatible, API-relevant few feature) => ([X] is incremented by one & [Y],[Z] are reset to zero)

Where things go wrong

Forget UML for documentation

- Use them for ad-hoc UML diagrams for a meeting, or informal UML Diagram for a high-level tutorial
- Don't consider them as central part of the documentation

Forget MS Word or HTML for documentation (if you can)!

- Keep your documentation close to the code
- Optional: auto-generate external documentation
- Beware: documents are almost impossible to keep in-sync with the "real truth"

Forget writing documentation early

- Don't spend all the time thinking about how to externally link class A with class B and algorithm C

Forget Documenting boilerplate code

- Ex Getter and setter.
  - If these 2 functions don't do more than getting and setting, then don't document it

Forget documenting trivial code

- Don't do everything (like every single line of code)

TL;DR: Keep things simple and concise

Create good documentation:

- By keeping documentation simple & concise
- By keeping documentation close to the code and close to the API, which are the ultimate truths of your application
- By keeping your documentation DRY
- By making documentation available to others, through a ticketing system, version control, sematic versioning.
- By referencing ticket IDs throughout your available media.
- By forgetting about "external" documentation, as long as you can.

A beginner's guide to writing documentation

Why write docs

You'll be using your code for 6 months

- You might think every code you wrote during that moment make sense and require no documentation.
    - Your wrong, Dead wrong
        - After 6 months, you will forget why you wrote it that way
- Therefore, make sure you explain why you wrote/implemented that way

You want people to use your code

- People need to understand how your code will help them solve their problems
- They need to understand your code's purpose and implementation
- (people don't know why you project exists) => (they won't use it)
- (people can't figure out how to install your code) => (they won't use it)
- (people can't figure out how to use your code) => (they won't use it)

You want people to help out

People will help contribute to your code if they can understand how your code works through good documentations

You only get contributions after you have

- Put in a lot of work
- users
- Documentation

Documentation provides a platform for you first contribution

- When people start contributing to an open-source project, the first contribution they do is to help edit documentation.
    - Edit documentation is less scarier than changing code.

You want your code to be better

Writing documentation can help improve the design of your code

- It allows you to brainstorm your API & design decision in a more formalized way
- Allows people to contribute to your code that follows your original intentions as well

You want to be a better writer

- Writing documentation is a useful skill as a programmer
- Keep your documented will keep you writing at a reasonable cadence
- Start simple like have a simple idea and an outline at first, then expand from them.

Version controlled plain text

Basic Example

- Online documentation: http:/.....
- Conference: http://.....

README

- Code hosting services will render your README into HTML automatically
- It's the first interaction that most users will have with your projects

What to write

- First appeal to 2 audiences: Users and Developers

What problem your project solves

Explain what your project does and why it exists

A small code example

Show example of what your code would normally be used for

A link to your code & issue tracker

There are people who want to report bugs and issues from your code

Make sure to make it easier for them to contribute

FAQ

If a lot of people have the same problem, then you will have to fix your documentation

Make sure to keep your documentation up to date

How to get support

Create a FAQ page for user to check out the most common problems

Info for people who want to contribute back

Document your standard on how you expect things to be done in the projects

Installation instructions

Your install instruction should be a couple line for the basic case

A page that gives more information should be linked from here if necessary

Your project's license

Types of license: BSD, MIT, GPL

- It's important to show your project's license for those who want to use your code

Template

Choose a simple template to follow for your README

Name the file README.md if you want to use markdown, or use README.rst if you want to use reStructureText

Core Practices for Agile/Lean Documentation

1. Writing
    1. Prefer executable specifications over static documents
        i. Executable specification
            - Information captured in traditional documents
                1. Requirements specification
                2. Architecture specification
                3. Design specification
        ii. Test-Driven Development (TDD) approach
            - Write detail specification in a just-in-time (JIT) basis
            - Before writing sufficient functionality to fulfill the test, you write test either at the customer/acceptance level or the developer level
            - Tests serves for 2 purposes:
                1. Specify the requirements/architecture/design
                2. Validate your work
    2. Document stable concepts, not speculative ideas
        i. According to figure 1, agile strategy is to defer the creation of all documents as late as possible, creating them just before you need them via a practice ("document late")
            1. Ex. System overview are better to write at the end of the development
                1. Also majority of user and support documentation is better written towards the end of the lifecycle.
                2. You can still take some point form notes along the way

ii. Waiting to document information when it's stabilized can help you reduce
- Cost - Don't waste time documenting info that changes
- Risk - less chance that your existing documentation will be out of date

iii. Disadvantages of this approach
- Forgotten some of the decisions behind the scenes
- May not have the right person anymore to write it b/c they got moved on to the new projects
- May not have funding to do the work
- The will to write the documentation may no longer exist

3. Generate system documentation
   i. We can save more money by generating the majority of the system documentation that we need

2. Simplification
   1. Keep documentation just simple enough, but not too simple
      i. Don't
         1. Create a 50-page document when you can do 5 pages
         2. Create 5 pages document when you can do bullet points
         3. Create an elaborate & intricately detailed diagram when a sketch can do
         4. Repeat info found elsewhere when a reference will do

   2. Write the fewest documents with least overlap
      i. Build larger documents from smaller projects
         1. Ex. When documenting in HTML, focus on one topic in one page
         2. Advantage
            1. Certain information is defined in only one place

   3. Put the information in the most appropriate place
      i. Understand the needs of the customers of certain information
         1. Where would they need certain info
      ii. Record the info where you enhances your work the most
      iii. Consider these issues when writing issue
         1. Indexing
         2. Linking
         3. accessibility

   4. Display information publicly
      i. It's a way of transferring information and communication through application
      ii. The greater the communication on your project the less need for detailed documentation

3. Determining What to Document

   1. Document with a purpose
      i. Only create document if it fulfills a clear, important and immediate goal of your overall
      ii. These purposes may by short or long term
      iii. Each system has its own documentation needs

   2. Focus on the needs of the actual customers of the document
      i. Work closer with the audience, so you will know what is important to be implemented in the documents
      ii. To do:
         1. Identify who the potential customer of your documentation
            1. What they believe they require

2. Negotiate with them the subset that they need
   iii. Discover what they believe by asking:
      1. What they do?
      2. How they do it
      3. How they want to work with the documentation

3. The customer determines sufficiency
   i. This practice provides a fair and effective quality gate between developers and the customers of our work
   ii. As writer of the documentation, it is your job to ensure that it has true meaning and provides value to your customer

4. Determining When to Document

   1. Iterate, iterate, and iterate
      i. Create documentation throughout the entire software development lifecycle (SDLC)
      ii. Write a little bit, show it to someone, get feedback, and then iterate

   2. Find better ways to communicate
      i. The #1 issue with communication is one of understanding and not of documentation
      ii. Goal:
         1. ensure maintenance developers understand how the system works so they can evolve it over time
         2. Enable your support & operations staff, not bury them with paper
      iii. Alternative than documents: one-on-one meetings

   3. Start with models you actually keep current
      i. If you chose to keep your diagrams up to date throughout development, then it's a good sign that these are valuable models that you should base your documentation around

   4. Update only when it hurts
      i. Many times, a document can be out of date and does matter that much

5. General
   1. Treat documentation like a requirement
      i. It should be prioritized
      ii. Any investment you make in documentation is investment that you could have made in new functionality
      iii. You shouldn't create documentation because your process says you should but because your stockholders say you should

   2. Require people to justify documentation requests
      i. When you explore documentation issue with your project stackholders, you will discover that they're asking for it because they don't trust you
         1. They don't understand the implication of what they're asking for
      ii. Better to ask them what they intend to use the documentation for and how they actually use the documentation

      iii. Important issues
         1. You should understand the total cost of ownership (TCO) for a document, and strive to maximize stakeholder ROI to provide the best value possible to your organization
         2. Someone must explicitly choose to make the investment in the documentation

3. The benefit of having documentation must be greater than the cost of creating and maintaining it
4. Ask whether you NEED the documentation, not whether you want it.

3. Recognize that you need some documentation
   i. Documentation is a part of the system as source code
      1. Sometimes you need to deliver user manuals, support documentation, operation documentation, and system overview documentation
   ii. Your teams' primary goal is to develop software, and the secondary goal is to enable your next effort
      1. Ensure that people who come after you can maintain and enhance it, operate it, and support it is also important
   iii. Documentation still serves a purpose
      1. Capture high-level information
      2. Need to capture important information
   iv. Agilists are in fact writing documentation
      1. Agile teams were just as likely as traditional teams to write deliverable documentation like manuals, operations documentation, and etc

4. Get someone with writing experience
   i. Strategies if you cant find one:
      1. Consider reading & following the advice presented in UnTechinical writing or taking a night-school course in writing fundamental
      2. Try writing documentation with a partner like there is significant value pair programming there is similar value in "pair documenting".
      3. Have shared ownership of all documentation so that multiple people will work on it.
      4. Purchase text-to-speech software that allows you to listen to what you written, a great way to discover poorly written passages.